INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

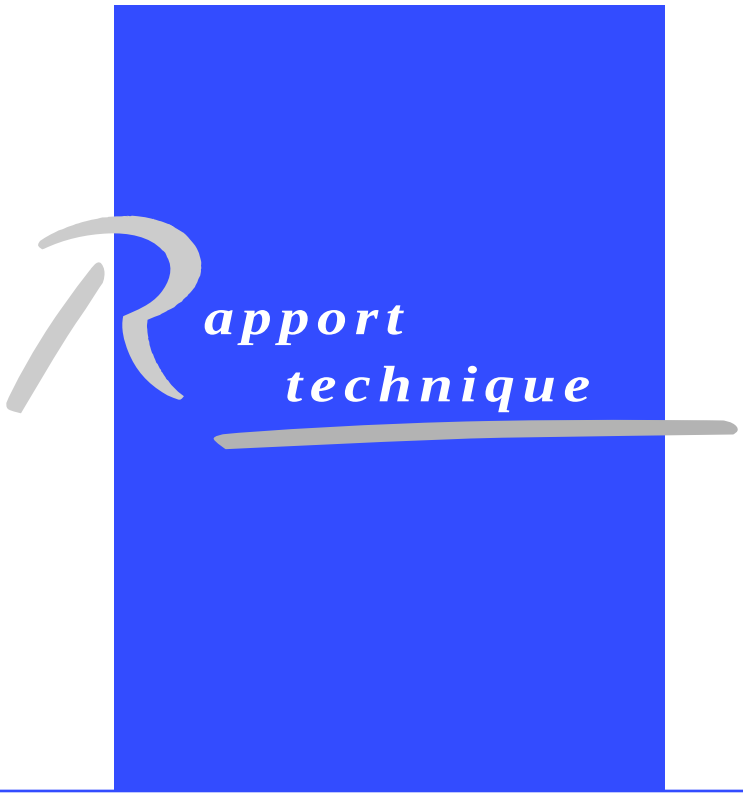
Zero Overhead Java Thread Migration

Sara Bouchenak and Daniel Hagimont

N° 0261

Mai 2002

THÈME 1



ISSN 0249-0803

Zero Overhead Java Thread Migration

Sara Bouchenak and Daniel Hagimont

Thème 1 — Réseaux et systèmes
Projet Sardes

Rapport technique n° 0261 — Mai 2002 — 33 pages

Abstract: The wide diffusion of Java is partly due to its mechanisms for mobile computing. Java provides most of the functions required to implement mobile applications, essentially code mobility (i.e., dynamic class loading) and data mobility (i.e., object serialization). However, Java does not provide any mechanism for the mobility of the computation (i.e., threads migration). Several projects have addressed the issue of Java thread migration, e.g., Sumatra, Wasp, JavaGo, Brakes, CIA. Most of these projects have attempted to minimize the overhead incurred by the migration mechanisms on applications performance, but none of them has been able to completely avoid this overhead. We propose a Java thread migration mechanism that does not incur any performance overhead on migratory threads. In this report, we present the design choices which allowed us to cancel the performance overhead, and we describe the implementation of our thread migration prototype in Sun Microsystems' JDK. We report on the experimental results that confirm the elimination of the performance overhead.

Key-words: threads, migration, performance, type inference, dynamic deoptimization, Java, virtual machines

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Migration de threads Java sans surcoût

Résumé : Le large succès qu'a connu Java est en partie dû à ses mécanismes de mobilité, tels que les outils de mobilité du code (chargement dynamique de classes) ou les fonctions de mobilité des données (la sérialisation d'objets). Cependant, Java ne fournit pas de mécanisme pour la mobilité de l'exécution (migration de threads). Plusieurs projets se sont récemment intéressés au problème de migration de threads Java, tels que Sumatra, Wasp, JavaGo, Brakes et CIA. La plupart de ces projets proposent des solutions qui minimisent le surcoût induit par la migration sur les performances des applications migrantes, sans annuler ce surcoût. Nous proposons un mécanisme de migration de threads Java qui n'induit aucun surcoût sur les performances des threads migrants. Dans ce rapport, nous présentons les choix de conception qui nous ont permis d'annuler le surcoût et décrivons la mise en œuvre de notre prototype de migration de threads dans le JDK de Sun Microsystems. Nous présentons également les résultats de l'évaluation de notre mécanisme, résultats qui confirment l'élimination de tout surcoût sur les performances des threads migrants.

Mots-clés : threads, migration, performence, inférence de types, désoptimisation dynamique, Java, machines virtuelles

1 Introduction

With the development of large scale systems (e.g., the Internet) and mobile users, many researchers in operating systems have investigated the design and implementation of mobile object systems. The overall goal is to be able to move computation between distributed machines, in order to balance the workload, to co-locate computation with data on remote machines or to follow a user in his displacements. Many prototype systems have been implemented, addressing some or all of the following issues:

- Moving code components. The system should allow code components to be remotely installed on machines and it should allow dynamic binding between legacy and dynamically installed components. Sun Microsystems' Java Applets [29] and Microsoft's ActiveX [1] are examples of such mobility.
- Moving data objects. The system should allow complex structures of objects to be moved between machines while keeping a consistent view of the overall object universe. Most mobile agent platforms provide such functionality, e.g., IBM's Aglets [15], Mole [4] or Concordia [31].
- Moving processes. The system should allow process migration, allowing a thread interrupted on one machine to resume its execution at the same point on another machine. Emerald [17], Sprite [10]and Charlotte [3] are distributed systems providing process migration facilities.

Traditionally, computing environments are heterogeneous, consisting of a mix of different hardware and operating systems. The advent of Java [26], which provides the abstraction of a homogeneous environment, led to a multitude of middleware environments which exploit, for different purposes, object, code or process mobility. Among them, we mention mobile agent systems [20], meta-computing environments [7], dynamic Web browsers and more generally dynamic reconfiguration of distributed systems [13].

Java provides most of the functions required to implement such middleware environments. Indeed, Java provides an object serialization mechanism which allows the capture and restoration of objects' state and therefore the migration of objects between several machines. Java also allows classes to be dynamically loaded and therefore to be moved between several nodes.

However, Java does not allow threads to be migrated. Thread migration consists in capturing the current execution state of the thread (mainly the thread stack) and then transferring this state to a target node on which execution is restored. But the standard Java API does not provide access to the execution state of Java threads, this state remains inaccessible to Java programmers.

1.1 Context of our research

Because the execution state of a Java thread is internal to the Java virtual machine (JVM) and is not directly accessible to Java programmers, the most intuitive approach is to add

new functions to the Java environment in order to export the thread state from the Java virtual machine. This solution grants full access to the entire state of a Java thread, and has been chosen by several Java thread migration projects such as Sumatra [2], Merpati [24] and CIA [16]. The main drawback of this approach is that it depends on a particular extension of the Java virtual machine, and the thread migration mechanism can therefore not be used on existing virtual machines.

In order to address the issue of the portability of the migration mechanism on multiple Java environments, some projects, like Wasp [12], Brakes [30] and JavaGo [22], propose a solution at the application level, without relying on an extension of the JVM. This approach is based on a pre-processor that is applied to the code of the application prior to execution; this pre-processor transforms the code in order to add statements which capture and restore the state of the thread. The key advantage of application-level implementations is the portability of their migration mechanisms to all Java environments. However, they are not able to access the entire execution state of a Java thread, because this state is internal to the JVM. The resulting migration systems are therefore incomplete.

Whatever the level of implementation (JVM-level or application-level), all the existing solutions impose an important performance overhead on migratory threads. For instance, the JVM-level based systems suffer from inducing a significant overhead on thread performance (+335%, +340%) because some of them extend the Java interpretation process and none of them support Java execution optimization (JIT compilation). The application-level based systems impose a non negligible performance overhead (+88%, +250%) due to the statements added to the original code of the thread.

To summarize, all existing solutions to Java thread migration are characterized by three properties:

- the *completeness* of the accessed thread state,
- the *portability* of the mechanism across different Java environments,
- and the *efficiency* of the migration mechanism, i.e., its impact on the performance of thread execution.

Regarding the existing solutions, the migration systems based on a JVM-level implementation verify the completeness requirement but they lack in efficiency and portability. And the migration systems proposed at the application-level are portable but they are neither efficient nor complete.

1.2 Objectives

One of the first criticisms addressed to Java was its poor performance. An important effort was therefore made by Java/JVM's designers in terms of execution optimization and Just-In-Time (JIT) compilation which led to today's efficient JVM. Consequently, providing a new feature which incurs an important overhead on applications execution is simply not acceptable. In other words, in order to make a thread migration mechanism widely accepted

in the Java community, it must not degrade the performance of the applications that use it. Therefore, our primary objective has been to provide a mechanism which does not impose any overhead on thread execution.

Previous works on Java thread migration do not tackle the overhead feature but rather give the priority to the cost of the migration operation. These works are dedicated to applications that perform frequent migration operations such as mobile agent based applications. We chose to address the overhead feature and thus provide an effective solution for applications in which migration is necessary but occurs rarely, such as the administration of distributed systems and more generally the dynamic reconfiguration of distributed applications.

Our second important objective is to provide a complete mechanism that captures the complete execution state of a Java thread.

Regarding portability, from our point of view, this property was not the main issue. Our approach was to give ourselves the opportunity to propose a complete and efficient Java thread migration system that would be widely used and could become a standard Java feature in future JVM implementations (this is what happened to RMI).

1.3 Our approach

The efficiency and completeness objectives led us to opt for a JVM-level solution. Indeed, this is the only way, on the one hand, to be granted access to the whole execution state of Java threads (completeness requirement), and on the other hand, to propose a solution without any performance overhead on migrating threads (efficiency requirement).

The main issue is to avoid the overhead on code execution, an overhead that all existing thread migration implementations incur. In the application-level solutions, this overhead is due to the statements added by the pre-processor, while in the JVM-level implementations the overhead has two sources: the extension of the Java interpreter and the non-compliance with JIT compilation.

In order to allow a Java thread to migrate between heterogeneous environments, in a portable manner, the data representing the thread's internal execution state must be translated to a portable format. In order to do this, many projects propose to extend the Java interpreter for inferring the type information necessary for the translation internal-format/portable-format. We circumvent this problem by proposing a type inference system which is totally separated from the Java interpreter and which is only invoked at migration time. This type inference is discussed in section 3.

Moreover, the thread migration projects that are based on an extension of the Java interpreter are not compatible with Java JIT compilers, as JIT compiled code no longer uses the Java interpreter. For this reason, our proposal does not modify the Java interpreter. But because the thread migration mechanism needs to access the thread's execution state as it is managed by the interpreter (a portable format of the thread's state) and because this state may no longer be available due to JIT compilation, we use a de-optimization system which permits reversing the JIT compilation process and recovering the thread's state as managed by the interpreter. The de-optimization system is the subject of section 4.

To summarize, the general pattern of our approach is:

- not to modify in any way the execution engine used by Java applications (Java interpretation or JIT compilation)
- to only add the necessary operations that will be executed at migration time (de-optimization, type inference, thread state capture, transfer and restoration).

1.4 Status

We extended Sun Microsystems' Java virtual machine, in its version Java 2 SDK/1.2.2 (formerly known as JDK 1.2.2). In this report, we describe the design and implementation of the extended JVM that support thread migration with the following properties:

1. The Java language syntax is not modified.
2. The Java compiler is not modified.
3. The existing Java API is not affected.
4. A new Java API for thread migration is proposed.
5. The implementation of the thread migration facility is based on type inference and dynamic de-optimization techniques.

Our API for thread migration is accessible at the following URL:

http://sardes.inrialpes.fr/research/JavaThread/api/

It consists of a Java package which provides two main facilities, the first allowing a Java thread to be captured/checkpointed (stored in a Java object) and restored/recovered (from this Java object), and the second facility allowing a thread to be migrated between two JVMs.

The implementation of our thread migration prototype comprises three main modules which respectively implement the type inference system, the dynamic de-optimization system and the thread state capture, transfer and restoration mechanisms (the three basic mechanisms for thread migration operations). This extension represents about 2.500 lines of Java source code and 17.500 lines of C source code.

Our prototype is freely available at the following URL:

http://sardes.inrialpes.fr/research/JavaThread/

It has been successfully used in the Suma metacomputing platform as a basic system for the implementation of a checkpointing/recovery facility for parallel computations [9]. In addition to Suma's designers, there were about 200 downloads from users, testers, students and researchers all around the world.

1.5 Roadmap

The rest of the report is structured as follows. Section 2 first presents the Java Virtual Machine's characteristics which are necessary to understand the rest of the report; it then describes the main issues and design choices which led us to our implementation. Section 3 focuses on the implementation of the type inference system which allows inferring the type of the information representing the thread's execution state. Section 4 describes how we used de-optimization techniques in order to comply with JIT optimizations. Section 5 discusses the related work. Sections 6 and 7 are respectively devoted to our performance evaluation and conclusions.

2 Overall design

We first present the main characteristics of a Java virtual machine. We then present the overall design of our Java thread migration implementation.

2.1 JVM characteristics

The Java Virtual Machine is the runtime environment on which applications developed in the Java object-oriented language can run. A program developed in Java is generally compiled in order to generate bytecode, a binary format which can be interpreted by a JVM. As the JVM is ported on most contemporary machines, a bytecode compiled Java program is portable between these machines.

The architecture of the Java environment is illustrated in Figure 1 where the JVM is presented as an abstraction of a homogeneous machine with a defined set of instructions, an execution engine (an equivalent of a hardware processor) and runtime data areas used for the memory and process management. In the following, we detail the bytecode properties we are interested in, the operating principles of the execution engine and the runtime data areas of the JVM.

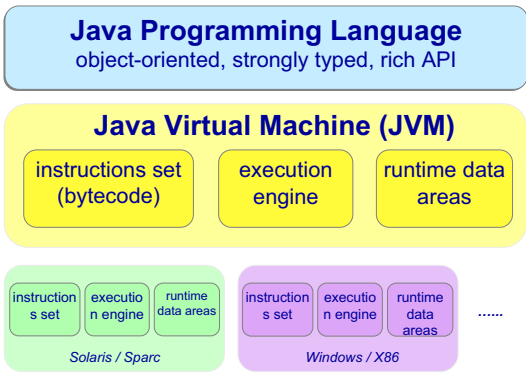


Figure 1. Architecture of the Java environment

2.1.1 Bytecode properties

The Java bytecode provides an instruction set that is very similar to the one of a hardware processor. Each instruction specifies the operation to be performed, the number of operands and the types of the operands manipulated by the instruction. For example, the bytecode instructions *iadd*, *ladd*, *fadd* and *dadd* apply respectively on two operands of type *int*, *long*, *float* and *double*, and return a result of the same type.

The execution of bytecode in a Java Virtual Machine is based on a stack. Figure 2 illustrates the execution of the *iadd* bytecode instruction which adds two integer operands.

Before the operation is invoked, the two operands *val1* and *val2* are pushed on the stack. After the operation is completed, the result *res* is left on top of the stack.

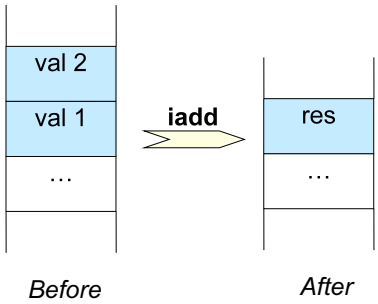


Figure 2. Addition of two integers in the JVM

2.1.2 Execution engine

The first generation of JVM was based on an interpreted scheme in which the interpreter translates each bytecode instruction into the execution of native code.

In order to improve performance, the second generation of JVM has integrated Just-In-Time (JIT) compilation, which compiles each method into native code at first call. Any subsequent invocation uses the compiled native version of the method, and therefore performs much faster (close to the performance of native code). However, if a method is rarely invoked, it is not worth compiling it at first call. This led to the introduction of adaptive JIT compilation, where only frequently invoked methods are dynamically compiled.

2.1.3 Runtime data areas

The Java virtual machine specification defines several runtime data areas [18]. Here, we focus our attention on the data areas describing the execution state (or execution context) of a Java thread.

The state of a Java thread is illustrated by Figure 3, it consists of three main data structures:

- *The Java stack.* A Java stack is associated with each thread in the JVM. The Java stack consists of a succession of frames. A new frame is pushed onto the stack each time a Java method is invoked and popped from the stack when the method returns. A frame includes two main data structures: a table containing the local variables of the associated method and an operand stack that contains the partial results (operands) of the method (e.g., the result of an *iadd* operation). The values of local variables and operands may be of several types: integer, float, Java reference, etc. A frame also contains registers such as the program counter and the top of the stack.

- *The object heap.* The heap of the JVM includes all the Java objects created during the lifetime of the JVM. The heap associated with a thread consists of all the objects used by the thread (objects accessible from the thread's Java stack).
- *The method area.* The method area of the JVM includes all classes (and their methods) that have been loaded by the JVM. The method area associated with a thread contains the classes used by the thread (classes where some methods are referenced by the thread's Java stack).

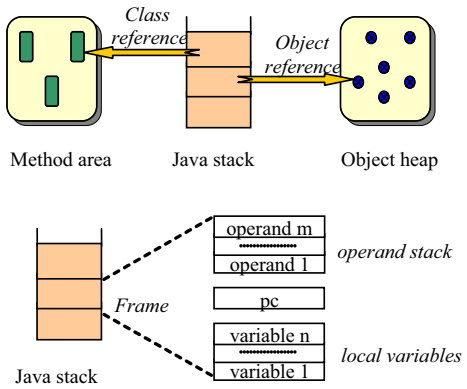


Figure 3. Java thread state and Java frame

To summarize, the execution state of a Java thread is composed of the three following data structures: the *Java stack* associated with the thread, a portion of the *object heap* including all the objects used by the thread and a portion of the *method area* including the classes used by the thread.

Notice that the Java stack is managed for the execution of a bytecode by a thread, i.e., when the underlying execution engine is a Java interpreter. But when a method is dynamically compiled by a JIT compiler, the invocation frames of this method are not managed on the Java stack anymore, but on a native stack associated with the thread.

2.2 Main issues and design choices

Our main objectives have been to focus on the completeness and the performance of the migration mechanism. First, regarding the completeness objective, in order to be able to capture the entire state of a thread, we decided to implement the mechanism within the JVM. This is the only way to be granted full access to the stack of a Java thread.

However, even if the data structure representing the stack of a Java thread is accessible within the JVM, this data structure cannot be transferred to another distant JVM as is. These data structures may include integers, booleans or Java references, which have to be consistently marshalled and unmarshalled before and after the effective transfer. But a

JVM does not manage any type information about the data stored in a Java stack, because a thread is considered as a native object which is not supposed to be moved between machines. Therefore, the difficult issue here is to infer the types of these data stored in a Java stack.

The only place where these types are known is the bytecode of the methods that push the data on the stack. As explained in section 2.1.1, a bytecode instruction which pushes a value on a Java stack is typed and determines the type of this value. The simplest solution is to modify the Java interpreter for each bytecode instruction and to store the type information "somewhere", when a value is pushed on the stack by this instruction (this is equivalent to managing a separate type stack for each thread). However, this solution has several important drawbacks. It requires significant modifications to the JVM and it introduces an important overhead on the bytecode execution, since the type stack has to be managed. As explained in section 1.2, an important objective is to avoid any overhead on the execution of applications.

In order to avoid any overhead, type inference must be performed at migration time. We have designed a solution in which we analyze the bytecode of the application at migration time in order to retrieve the type of the stacked data. We show in section 3 that it is possible to retrieve the type of all the stacked data with one pass on the application bytecode. With this technique, the Java interpreter is kept unchanged and no overhead is incurred.

A second performance requirement is compatibility with today's JVM JIT-compilation techniques. The problem is that the invocation of a JIT compiled method is pushed on the native stack (on which the Java interpreter runs), which is separate from the Java stack. However, we observed that in one of the JVM implementations proposed by Sun Microsystems (JDK 1.3.1), an internal interface allows building a Java method invocation frame (the same as would have been pushed on the Java stack by the interpreter) from the frame on the native stack (pushed by the JIT compiled method code). We describe in section 4 how we use this JVM internal interface in order to allow thread state capture, even in the case where JIT compiled method invocations are in the thread invocation stack.

3 Type inference

This section first describes the technique used to retrieve the types of all data on a thread's Java stack in one pass of the bytecode, and then gives the current implementation status of this technique.

3.1 Design principles

At migration time, the capture of the thread state requires to get the type of the data included in the stack. As explained above, an important requirement is not to place any overhead on thread execution. Extending the Java interpreter for managing data types in the Java stack would incur such an overhead[1]. The type inference approach described in this section only adds computation at migration time. It consists of analyzing the bytecode of the application at migration time in order to retrieve the type of the stacked data. The algorithm of the bytecode analysis for type inference is partly drawn from the bytecode verifier algorithm described in section 4.9.2 of the Java virtual machine specification [18].

Our type inference mechanism aims at building a *type stack* that reflects the types of the values contained in the thread's Java stack. Like the Java stack, the type stack consists of a succession of frames which we call *type frames* (see Figure 4). A *type frame* on the type stack is associated with each Java frame on the Java stack. A type frame contains two main data structures: a table that describes the types of the local variables of the associated method and an operand type stack that gives the types of the partial results of the method.

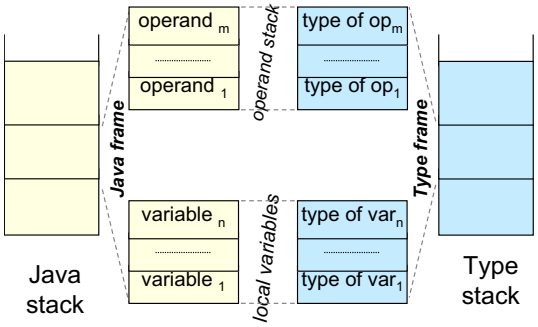


Figure 4. Type stack vs. Java stack

The type stack of a thread is built as follows. For each frame on the thread's Java stack, the bytecode of the associated method is parsed from the beginning to the exit point of the method (the migration point for the top-most frame or a nested method invocation for the

[1] Our first solution was based on an extension of the Java interpreter, i.e., ITS (Interpreter-based Thread Serialization) [5].

other frames). The objective here is to associate a type with each value (local variable or partial result) in the Java frame, and to store this type in the associated type frame. As explained above, the types of the values are easily inferred from the bytecode instructions.

The main problem when parsing the bytecode of a method is to determine the path that should be followed between the beginning of the method and the exit point, in the case where several paths exist. This problem is solved thanks to two correctness properties of the Java code [11]:

Correctness properties:
At any given point in the program, no matter what code path is taken to reach that point:
P1: The operand stacks built by following each code path contain the same types.
P2: The local variables built by following each code path are of the same types or are unused if the types differ.

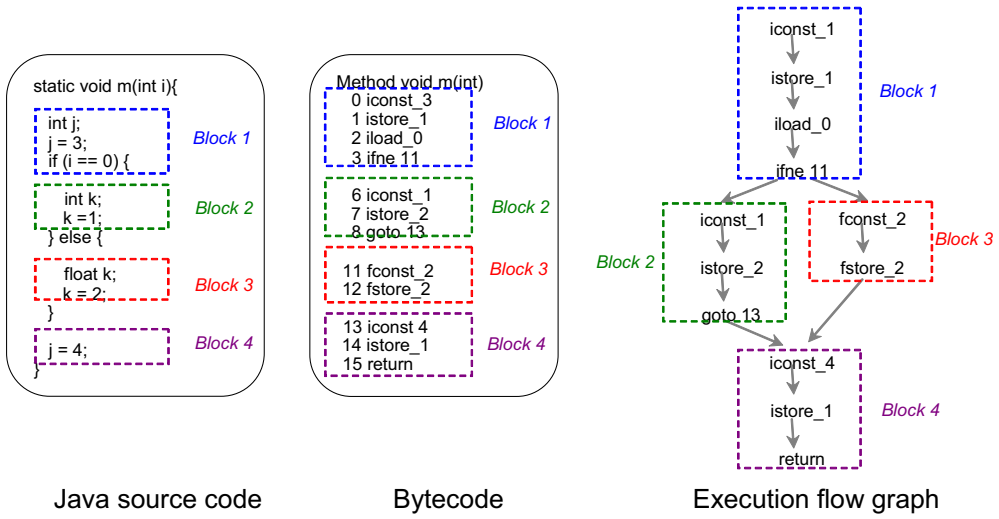


Figure 5. Example of bytecode execution

Figure 5 presents an example of a Java program illustrated by the Java source code, its bytecode equivalent and the associated execution flow graph. In this program, the local variable *k* is declared of type *int* in *block 2* and of type *float* in *block 3*. The *k* variable is

implemented by the same memory word in the Java frame (a variable of index 2, manipulated at lines 7 and 12 in the bytecode). Therefore, for an exit point in *block 2* or *block 3*, the type of this memory word in the Java frame will respectively be *int* or *float*. If the exit point is in *block 4*, the type of this memory word is undefined because no variable is associated with it at the exit point (k is not accessible from *block 4*). Consequently, if the exit point is in *block 1*, *block 2* or *block 3*, then this point determines the path to be followed by our type inference algorithm (because there is only one path which reaches that point). If the exit point is in *block 4*, then no matter the path which is used by our type inference algorithm to reach that point.

The algorithm has to try different paths to reach the exit point; it memorizes the paths to be visited whenever a branch instruction is parsed. For each parsed instruction, if an operand is pushed or a variable assigned in the Java frame, the type is pushed or assigned in the type frame. If an already visited instruction is reached, then the algorithm gets back to another path among the memorized (not yet visited) paths in order to find one which will reach the exit point. When the exit point is reached, the type frame gives the types of the values on the Java frame.

To summarize, the determination of the types of the values on a thread's Java stack amounts to determining, for the code of each method currently executed by the thread, a code path starting from the beginning of the method's code and reaching the method's exit point, and inferring the types of the manipulated values from the bytecode instructions contained in this path. We implemented an algorithm that performs this in one pass of the bytecode. Our algorithm builds, at capture time, a type stack that reflects the types of the values on the thread's Java stack.

3.2 Implementation status

This type inference system has been implemented and integrated in our thread migration service, in Sun Microsystems' JDK 1.2.2. Our performance evaluation is based on this version (see section 6). We are currently porting this type inference system to JDK 1.3 (the HotSpot virtual machine in which the de-optimization functions are available) [19].

The algorithm of the type inference system has many similarities with the algorithm implemented by the bytecode verifier [11], which verifies at load time, the correctness of the dynamically loaded bytecode. We are currently working on the implementation of an optimized algorithm which should be cheaper in terms of memory and computation.

4 Dynamic deoptimization

4.1 Design principles

As motivated in section 1.2, one of our main objectives is not to trade Java applications performance for the implementation of thread migration. In particular, JIT compilation is the key technique which makes Java code execute close to the speed of native code. Consequently, the implementation of our migration service should be compatible with JIT optimizations.

With JIT compilers, Java methods are dynamically compiled to native code. This implies that the execution of these methods is no longer based on the thread's Java stack but on its native stack. However, the type inference technique described in the previous section requires access to the thread's Java stack. The issue is thus to permit thread migration even if the thread's Java stack does not reflect the execution of the Java methods that have been JIT compiled by the JVM's underlying execution engine. Here, we would need functions that allow us to restore Java frames from native frames produced by the JIT compiler, in order to be able to apply the type inference technique.

While examining the internal structures and interfaces of Sun Microsystems' Java virtual machine 1.3/HotSpot (JDK 1.3), we discovered that it includes a mechanism which performs dynamic deoptimization to transform of native frames associated with JIT compiled methods to Java frames [19]. Dynamic deoptimization was first used in the Self's source-level debugging system; it shields the debugger from optimizations performed by the compiler by dynamically deoptimizing code on demand [14].

Dynamic deoptimization was introduced in the HotSpot VM in order to deal with the inconsistency problem rising from the combination of method inlining performed by JIT compilation and dynamic class loading. Figure 6 illustrates this problem. In this example, a method *m1* calls a method *m2* of a class *C1*. For optimization purpose, the JIT compiler may inline *m2* in *m1*. But this inlining may become invalid if *C2*, a subclass of *C1* that overrides *m2*, is dynamically loaded. Here, dynamic deoptimization is used to revert from optimized (i.e., compiled/inlined) code to interpreted code.

Thus, dynamic deoptimization was used in the context of debugging systems and dynamic class loading systems. Here, we use it in a thread migration system. Indeed, at migration time, we invoke dynamic deoptimization on the thread's JIT compiled frames in order to retrieve the Java frames which would have been produced by the Java interpreter. Theredore, the type inference algorithm described in section 3 can be applied to these Java frames. After the thread migration operation is completed, the JIT compiler can of course perform new optimizations.

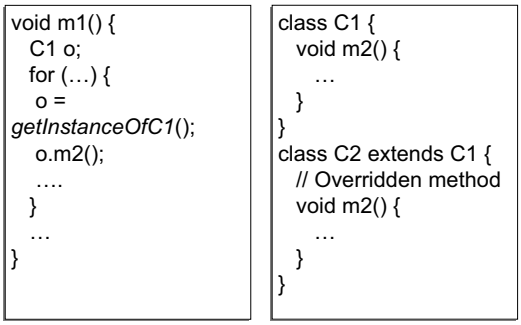


Figure 6. Method inlining and dynamic class loading

To summarize, a Java application executes exactly in the same conditions as on a non-extended JVM. The deoptimization mechanism permits reverting from native frames produced by JIT compiled methods to Java frames. Then, the type inference service is used to retrieve the type of the values managed in the Java frames.

4.2 Implementation status

We have experimented with the de-optimization functions and showed that we were able to retrieve the Java frames from the JIT compiled frames.

However, the de-optimization functions are only available in JDK 1.3 (HotSpot). We are completing the port of the type inference system (section 3) from JDK 1.2 to JDK 1.3 in order to produce an integrated prototype of our solution.

5 Related work

Systems providing Java thread migration can be implemented at different levels:

- the application level
- the virtual machine level.

The main motivation for implementing migration systems at the application level is to avoid the modification of the Java virtual machine. Such systems can therefore be used on any JVM implementation, i.e., Java threads can migrate between all existing and future JVM implementations. But this approach has two main drawbacks: i) it may incur a considerable overhead and, ii) it does not take into account the complete state of the migrating Java thread.

In contrast, migration systems implemented at the virtual machine level can solve the overhead problem. Moreover, because such systems are implemented at the virtual machine level, they can access the complete state of the migrating thread. However, providing a new JVM implementation requires that this JVM be installed on a machine before a Java thread can migrate to this machine.

5.1 Application-level approach

In the application level approach, the code (source code or bytecode) of the application is transformed by a preprocessor which adds new statements to the application's code. The added statements have two main goals:

- First, they attach a backup object to every Java thread that executes the application. This object is used to describe the execution state of the Java thread. The backup object is serializable; the associated thread can therefore be made mobile.
- Second, the statements added by the preprocessor manage the thread state capture and restoration operations. For each invoked method, this added code saves in the backup object the local variables of the method, the items of the frame's operand stack and the exit point in the method. Thus, at any execution point, the thread state is accessible in the backup object. The restoration is achieved by re-executing the modified version of the application code (produced by the preprocessor). This rebuilds the Java stack with the appropriate frames. The preprocessor inserts, at the beginning of each method, a code sequence which checks whether a restoration is in progress. If so, the local variables and the operand stack items are initialized with the values stored in the backup object, and execution jumps to the point where the method exits, according to the saved exit point.

Several Java thread migration systems follow this approach: Wasp [12] and JavaGo [23] provide a source code preprocessor while Brakes [30] and JavaGoX [22] are based on a bytecode preprocessor.

Two main techniques have been used to capture the thread state at the application level. In the first, which we call *control-return technique* (Brakes), the preprocessor inserts a particular code block into the application code after each method invocation, which checks whether the thread is in the capturing mode, and if so, first saves the state of the current frame, and then returns to the calling method. The use of *return* instructions allows control to be transferred back to the calling method and thus the state of the previous frame to be saved in its turn. This process is recursively repeated until the entire Java stack is parsed. The second technique, which we call *exception-based technique* (Wasp, JavaGo and JavaGoX) relies on exceptions to parse all the method frames on the stack. The capture operation is notified by throwing a particular Java exception that we call *NotifyCapture*. Each method of the application is transformed by the preprocessor in such a way that it catches the *NotifyCapture* exception and handles it by saving the current state of the method. The *NotifyCapture* exception is then propagated to the calling method, and so forth.

The main advantage of the application level approach is that it works with unmodified JVMs. Moreover, it is fully compliant with JIT compilers. However, it has two main drawbacks which make it hardly acceptable:

- the overhead on applications performance,
- the incompleteness of the solution.

Due to the new code inserted by the preprocessor in the original code of the application, a significant overhead may be incurred on applications performance (see section 6). Such an overhead may, for instance, reduce the benefit of using thread migration for dynamic load balancing in distributed systems.

Another drawback of the application level approach is that it does not allow capturing the entire state of the migrating Java thread. For instance, it is not possible to perform a migration during the execution of a *finally* clause. This is because this clause requires the manipulation of a *returnAddress* value, a memory address, which is not portable between several machines. Therefore, the Java thread migration systems implemented at the application level impose some constraints, such as not permitting thread migration in a *finally* clause.

Our implementation of thread migration allows capturing the entire state of a migrating thread and it does not impose any overhead on the execution of Java applications, the cost of the mechanism being paid only at migration time.

5.2 JVM-level approach

In JVM-level approaches, the JVM is extended in order to allow capturing and restoring Java thread states. Some JVM internal interfaces are sometimes reused to implement the mechanism.

In the CIA project [16], they proposed to reuse the Java Virtual Machine Debug Interface (JVMDI) to implement Java thread migration. The JVMDI is a programming interface used by debuggers and other programming tools [28]. It provides a way both to inspect the state

and to control the execution of applications running in the Java Virtual Machine. The main motivation for this solution is not to modify the JVM, by exploiting existing JVMDI features in order to inspect and control the state of Java threads.

For capturing the state of a Java thread, CIA uses some JVMDI's functions in order to parse the thread's Java stack, and to save, for each frame on the stack, the values of the local variables and a relative value of the program counter register[2]. For state restoration, CIA uses the callback mechanism provided by the JVMDI in order to catch events when methods are entered. The application to be restored is restarted and for each entered method, the callback re-initializes the local variables and the PC using the saved values. This process is recursively repeated until the interruption point at capture time is reached; at this point, the thread can resume its execution.

However, even if CIA's thread migration facility is provided at the JVM level, it does not avoid the two main drawbacks of application level solutions: neither the performance overhead nor the incompleteness. Indeed, because CIA is based on the JVMDI, the JVM running CIA must be launched in debug mode. This implies that Java JIT compilation is disabled and that an important overhead is imposed on the executed code (about +800%). Regarding the incompleteness of the solution, the JVMDI does not provide any way to access the operand stack items; therefore CIA imposes some limitations on the use of expressions at the application programming level.

To tackle the problem of inaccessibility of the complete state of Java threads, several projects (e.g., Sumatra [2], ITS - Interpreter-based Thread Serialization - [5], Merpati [24]) introduced extensions to Sun Microsystems' JVM, and more precisely extensions to the bytecode interpreter. As identified in section Main issues and design choices, the main problem is to get the types of the values on the Java stack. The proposed solutions rely on the typing of bytecode instructions in Java. It is therefore possible, during the interpretation of the bytecode instructions, to determine the types of the values manipulated by these instructions. The Java interpreter is extended to manage both the Java stack and a type stack which registers the type of the values stored in the stack. This solution presents the advantage of providing access to the entire state of a Java thread but it has two main drawbacks:

- The performance overhead imposed by this technique is significant, since it impacts most of the bytecode instructions for managing the type stack.
- It is not compatible with JIT compilation because JIT-compiled code is no longer interpreted whereas the proposed technique is based on an extended interpreter.

To summarize, all the previously proposed solutions have serious drawbacks. First, except for the solution based on an extension of the Java interpreter, none of them propose a complete thread state capture mechanism, which implies limitations at the application programming level. Second, all of them impose an important overhead on thread execution. JVM-level approaches are not compatible with JIT optimization and therefore per-

[2]The relative value of the PC allows the captured thread to be restored on a different machine.

form poorly. Application-level approaches are JIT compliant, but the code inserted by the preprocessor generates an important overhead.

Our contribution is to provide a solution which allows capturing the entire state of a Java thread, without adding any performance overhead on migrating threads; that is CTS (Capture-time based Thread Serialization).

6 Evaluation

This section presents some experimental results showing the efficiency of our Java thread migration system. It first outlines the objectives of this evaluation and the characteristics of the evaluation environment and then presents and discusses the performance results.

6.1 Objectives

In the evaluation of the thread migration mechanism, two measurements have to be reported (Figure 7):

- *The overhead on code execution.* The overhead on code execution is defined as the difference between the cost of code execution on the system which includes the migration mechanism (*E2a* + *E2b*) and the cost of code execution on the system which does not include the migration mechanism (*E1*), that is: *E2a* + *E2b* - *E1*
- *The latency of migration.* The latency of migration (*Lat*) is defined as the sum of thread state capture, transfer and restoration.

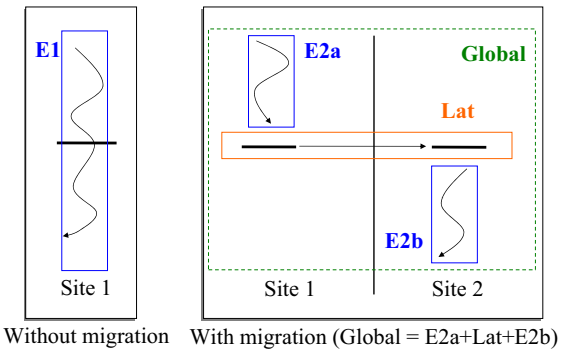


Figure 7. Performance overhead on code execution vs. latency of migration

Naturally, the overhead on code execution and the latency of migration both depend on the application on which it is measured. In particular, they depend on the content of the Java stack, e.g., the number of Java frames on the stack.

The objective of the evaluation presented in this section is to illustrate the two following properties:

- **The only system without performance overhead on code execution.** CTS is especially designed to avoid any change in the application bytecode, the bytecode

interpreter or the code generated by the JIT compiler. But this is not the case of all existing Java threads migration systems. Indeed, CTS is the only system without any performance overhead on code execution. Our performance evaluation aims at comparing this performance overhead for the different approaches.

- **Cost transferred to migration latency**. The cancellation of the performance overhead in the CTS system is not a miracle and it has its counterpart; it is obtained by moving all the costs of the migration mechanism from execution time (overhead on code execution) to migration time (latency of migration). Previous projects had a different balance between these two costs.

6.2 Evaluation environment

The performance results presented in this section were obtained in the following environment:

- Synesis, Pentium III, 1 GHz uni-processor, 256 MB RAM,
- Windows NT, SP 4,
- Sun Microsystems' Java Development KIT/Version 1.2.2, also known as Java 2 SDK/Version 1.2.2.

6.3 Performance overhead

Here, our objective is two-fold:

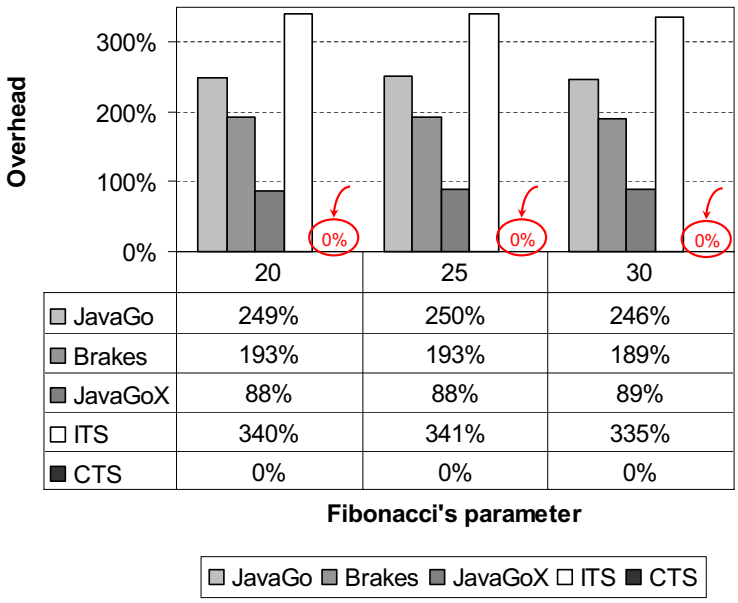
- to confirm what we explained in the previous sections, that is our CTS implementation of Java thread migration does not incur any performance overhead on code execution.
- to show that other implementations all incur a significant overhead.

In order to do this, we installed and configured several Java threads migration systems, which cover all the approaches to Java thread migration described in section 5:

- JavaGo [23], an application-level system based on a pre-processor of the application source code executed by the Java migratory threads,
- Brakes [30] and JavaGoX [22], two application-level systems based on a pre-processor of the bytecode executed by the migratory threads,
- ITS [5], the first solution that we proposed, based on an extension of the Java interpreter of the JVM,
- and finally CTS, our final solution, based on the type inference and the dynamic de-optimization techniques in the JVM.

We then used the Fibonacci algorithm in order to evaluate and compare the performance overheads incurred by the migration systems we installed. Fibonacci is a recursive algorithm which generates a lot of information on the Java stack, and it has been used as a benchmark for the evaluation of the overhead on code execution by many projects [16, 22, 23, 30]. The results are presented in Figure 8; this figure shows that:

- JavaGo , Brakes and JavaGoX incur a non negligible performance overhead (+88% to +250%), due to the code inserted by the pre-processor in the application code.
- ITS imposes an important overhead (+335% to +340%) due to the additional processing performed by the underlying extended Java interpreter.
- CTS does not incur any performance overhead, as discussed in the previous section.

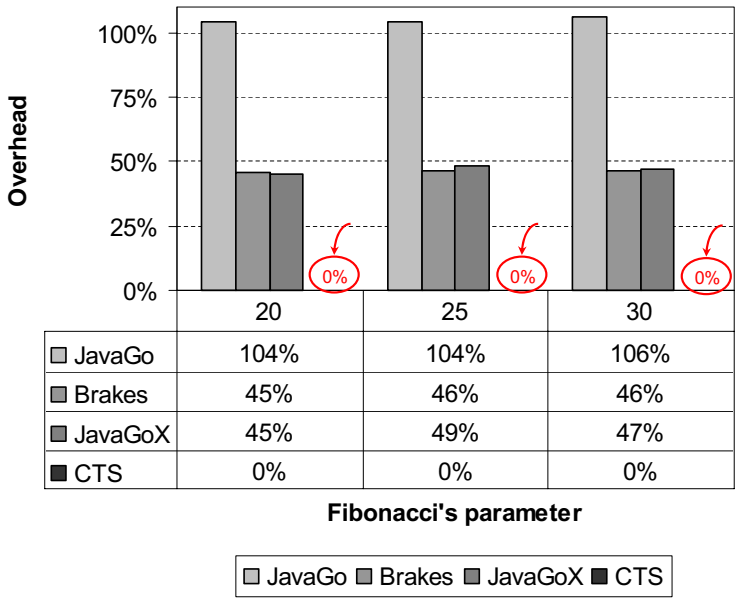


	20	25	30
JavaGo	249%	250%	246%
Brakes	193%	193%	189%
JavaGoX	88%	88%	89%
ITS	340%	341%	335%
CTS	0%	0%	0%

Figure 8. Fibonacci Benchmark/ JIT compilation disabled: Performance overhead

In Figure 8, the illustrated performance figures result from benchmarks running without Java JIT compilation. This was necessary in order to be able to evaluate the ITS system and compare it with other systems. ITS is indeed based on an extended Java interpreter and can therefore only be used in interpreted mode (without JIT compilation). But because of the generalization of the use of Java JIT compilers, the effective (with JIT compilation) performance overheads incurred by the other systems (JavaGo, Brakes, JavaGoX, CTS) are

presented in Figure 9. Finally, even if JIT compilation reduces the performance overhead incurred by JavaGo, Brakes and JavaGoX, it does not cancel it (+45% to +106%), thus heavily penalizing Java migratory applications.



	20	25	30
JavaGo	104%	104%	106%
Brakes	45%	46%	46%
JavaGoX	45%	49%	47%
CTS	0%	0%	0%

Figure 9. Fibonacci Benchmark/ JIT compilation enabled: Performance overhead

6.4 Cost transferred to migration latency

The result of the evaluation presented in the previous section is that CTS is the only Java thread migration system that does not incur a performance overhead on migratory threads. This behavior is not surprising; it is due to the fact that CTS does not perform any additional processing while the application executes. In other words, with CTS, there is no performance overhead because all additional cost is transferred to migration latency.

In this section, we aim at discussing the relationship between, on one hand, the variation of the performance overhead on a migratory thread and, on the other hand, the variation of the migration latency.

In order to be able to compare the latency of a Java thread migration using different migration systems, the environmental characteristics for the evaluation of the different systems must be as similar as possible. When studying the different thread migration implementations, we noticed that each migration system proposes its own implementation of the thread's state transfer, one of the three steps of a thread migration (i.e., state capture, transfer and

restoration). Different implementations of thread state transfer incur different state transfer times, and because most of the time of a thread migration is spent in the transfer of the state [6], the comparison between the different migration systems may be distorted. To homogenize the evaluation environment, we implemented a common state transfer mechanism, which is simply based on Java object serialization and system Java class loader. We then modified each migration system by re-implementing its transfer operation in order to make it use the new transfer mechanism.

Another aspect for the comparison of the latency of a Java thread migration using different migration systems is the homogenization of the benchmarks used for the evaluation of the migration latency with the different systems. Indeed, with all the migration systems, the migrating thread must have the same execution context when it migrates, i.e., similar Java frames pushed on its Java stack, similar Java objects in its heap. In order to achieve this in our evaluation, we fixed the number of Java objects used by the migrating thread and focused on the variation of the number of Java frames on the thread's Java stack. To vary the number of frames on the thread's stack, we wrote a benchmark program that first performs recursive calls of a Java method and, when it reaches the deepest recursive call, runs a ping-pong program using the migration facility of the underlying benchmarked system in order to measure the average latency of a migration operation.

For the evaluation of the migration latency with JavaGo, Brakes and JavaGoX, the benchmark program was written following the programming constraints of each system and then passed through the pre-processor before running it on the standard JVM. The benchmark program for ITS and CTS was written and run on the associated extended JVM.

On the other hand, in order to be able to illustrate the relationship between the variation of the migration latency and the variation of the performance overhead on a migratory thread, the two evaluations must be performed in a similar evaluation environment. And because the variation of the migration latency was evaluated according to the number of frames on the thread's Java stack, we also evaluated the variation of the performance overhead on the execution of a program according to the number of frames that this execution pushes on the thread's stack. For measuring the performance overhead, we used a benchmark program that performs recursive calls in order to be able to vary the number of frames.

For the evaluation of the performance overhead of JavaGo, Brakes and JavaGoX, the benchmark program was first written following the programming constraints of each system and then passed through the pre-processor before it was run on the standard JVM. Whereas for ITS and CTS, the benchmark program was written following the standard Java language syntax and run on the associated extended JVM (i.e., respectively ITS and CTS).

Figure 10 and Figure 11 illustrate, for the JavaGo , Brakes , JavaGoX , ITS and CTS systems, the variation of, on the left-hand side, the performance overhead on a migrating Java thread, and on the right-hand side, the thread migration latency. In Figure 10, the thread migrates when it has five Java frames on its Java stack and in Figure 11, the thread migrates when there are ten Java frames on its Java stack.

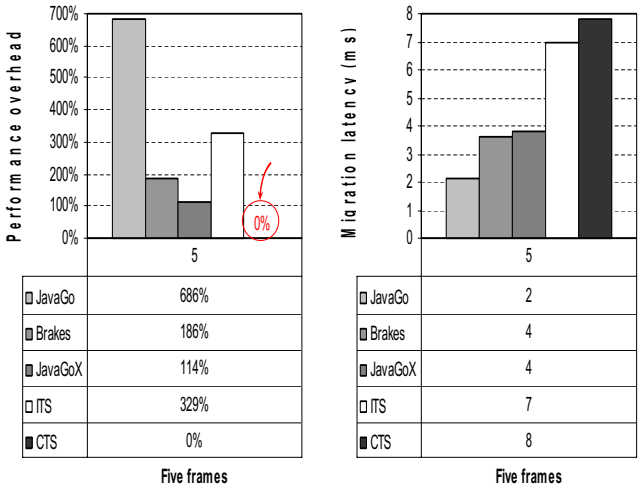


Figure 10. Performance overhead vs. migration latency, Thread migrating with 5 frames

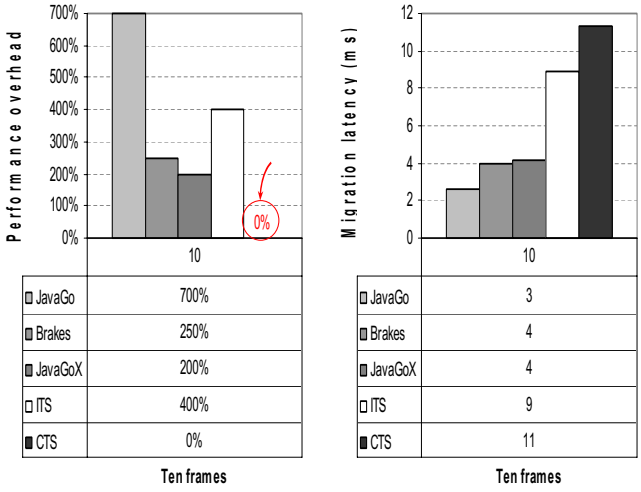


Figure 11. Performance overhead vs. migration latency, Thread migrating with 10 frames

The two last figures show that:

- For the application-level systems (JavaGo, Brakes, JavaGoX), the overhead on the migrating thread and the migration latency are inversely proportional. This is explained by the fact that the more processing is performed during the execution of the thread (performance overhead), the less processing is required at migration time (migration latency).
- Here, the performance overhead of ITS lies between the overhead of the Java source-level system (JavaGo) and the overhead of the bytecode-level systems (Brakes, JavaGoX). But the inverse proportion is not present in the migration latency; this is due to the fact that ITS meets the completeness requirement (see section Main issues and design choices) and thus performs more processing at migration time.
- CTS proposes the more expensive migration because everything is done at migration time.

6.5 Synthesis

To summarize, our evaluation experiments show that:

- Apart from the thread migration operation itself, the performance of the thread when it is running its own application code does not change. The inexistence of performance overhead with CTS may appear to be a general behavior but it is not. CTS is the only Java thread migration system without any performance overhead on migratory threads.
- The cancellation of the performance overhead in the CTS system is not magic; it is obtained by transferring all the additional cost to the migration latency.

As a result of the evaluation experiments, we can identify two kinds of behaviors in the existing Java threads migration systems:

- the migration systems that minimize the migration latency by adding an overhead on the migrating threads, this behavior is probably interesting for mobile agent based applications where migrations are frequent operations, and
- the migration systems that do not modify the "normal" performance of the migrating threads and put all additional cost in the migration operation, this kind of behavior targets the applications where the migration operations are necessary but occur rarely such as the administration of distributed systems.

7 Conclusion

Java provides most of the functions required to implement mobile applications, essentially code mobility (i.e., dynamic class loading) and data mobility (i.e., object serialization). However, Java does not provide any mechanism for the mobility of the computation (i.e., threads).

Several projects attempted to provide such a mechanism in the Java environment, but the proposed solutions were limited in terms of completeness - they don't allow capturing the complete state of a Java thread - and in terms of performance - they impose a significant overhead on applications performance.

In this report, we presented the design and the implementation of a Java thread migration mechanism which is optimal for both completeness and performance aspects. Our solution allows the capture and restoration of the complete state of a Java thread and it does not impose any runtime overhead on applications performance.

Our thread migration system has been implemented within the Java Virtual Machine, which grants access to the entire threads state. It relies on a type inference system which allows inferring the type of the information on a thread stack. This type inference system is totally separated from the JVM interpreter and does therefore not impact bytecode interpretation performance. In order to comply with JIT compilation, our migration system uses a de-optimization system which allows reversing the JIT compilation process. At migration time, native execution structures (resulting from JIT compilation) are de-optimized, which allows recovering the Java stack (as managed by the interpreter) and capturing it.

We reported on a performance evaluation based on our thread migration prototype and on several prototypes from projects which implemented different approaches. This evaluation demonstrates that we are able to have all the costs of the mechanisms paid at migration time, without any overhead on code execution. This is an effective solution for applications requiring a migration facility, with non frequent migration operations: the "normal" behavior of the applications must therefore be kept unchanged.

In this report, we described our work towards the provision of the basic mechanisms underlying the building of an effective and complete Java thread migration system. We restricted our discussion to the design and implementation issues in a local environment (i.e., a local JVM), and we did not discuss the problems rising from using our migration facility for building large distributed systems. Some elements of response are presented in [9], where the authors describe the building of a distributed fault tolerant system using our Java thread checkpoint/recovery mechanism. Indeed, Cardinale et al. describe their implementation of a checkpoint/restart facility for parallel computations in the Suma metacomputing system. The fault tolerance was the first experiment with our system, and in the future, we plan to experiment with our thread migration system to build large mobile distributed applications.

8 Acknowledgements

Our thanks to Christian Jensen, Sacha Krakowiak and Jacques Mossière for providing many useful suggestions that significantly improved this report.

References

[1] Aaron, B. Aaron, A., *ActiveX Technical Reference*, Prima Pub, April 1997.

[2] Acharya, A., Ranganathan, M., and Salz, J., Sumatra: A Language for Resource-aware Mobile Programs, *Mobile Object Systems: Towards the Programmable Internet, Lecture Notes in Computer Science*, Number 1222, April 1997.
http://www.cs.umd.edu/ãcha/

[3] Artsy, Y., Finkel, R., Designing a Process Migration Facility: The Charlotte Experience, *IEEE Computer*, Volume 22, Number 9, September 1989.
http://cs.engr.uky.edu/r̃aphael/

[4] Baumann, J., Hohl, F., Straber, M., Rothermel, K., Mole - Concepts of Mobile Agent System, *WWW Journal, Special Issue on Applications and Technologies of Web Agents*, Volume 1, Number 3, 1998.
http://mole.informatik.uni-stuttgart.de/

[5] Bouchenak, S., Hagimont, D., Pickling threads state in the Java system, *Proceeding of Technology of Object-Oriented Languages and Systems Europe - Europe (TOOLS Europe'2000)*, Mont Saint Michel / Saint Malo, France. June 2000.
http://sardes.inrialpes.fr/b̃ouchena/Publications/

[6] Bouchenak, S., Making Java Applications Mobile or Persistent, *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, San Antonio, Texas, USA, January 2001.
http://sardes.inrialpes.fr/b̃ouchena/Publications/

[7] Buyya, R., *High Performance Cluster Computing: Architectures and Systems*, Volume 1, Prentice Hall PTR, NJ, USA, 1999.

[8] Campadello, S., Koskimies, O., Raatikainen, K., Helin, H., Wireless Java RMI, *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC'00)*, Makuhari, Japan, September 2000.
http://www.cs.helsinki.fi/research/monads/

[9] Cardinale, Y., Hernández, E., Checkpointing Facility on a Metasystem, *European Conference on Parallel Computing (Euro-Par'2001)*, Manchester, UK, January 2001.
http://suma.ldc.usb.ve/

[10] Douglis, F., Ousterhout, J., Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software Practice and Experience*, Volume 21, Number 8, August 1991.
http://www.research.att.com/d̃ouglis/papers/

[11] Engel, J., *Programming for the Java Virtual Machine*, Addison Wesley, 1999.

[12] Fünfrocken, S., Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs), *Proceedings of Second International Workshop Mobile Agents 98 (MA'98)*, Stuttgart, Germany, September 9 - 11, 1998.
http://www.informatik.tu-darmstadt.de/~fuenf

[13] Hofmeister, C., Purtilo, J. M., Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement, *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, USA, May 1993.

[14] Hölzle, U., Chambers, C., Ungar, D., Debugging Optimized Code with Dynamic Deoptimization, *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, California, USA, June, 1992.
http://www.cs.ucsb.edu/~urs/

[15] IBM Tokyo Research Labs, *Aglets Workbench: Programming Mobile Agents in Java*, 1996.
http://www.trl.ibm.co.jp/aglets

[16] Illmann, T., Krueger, T., Kargl F., Weber, M., Transparent Migration of Mobile Agents Using the Java Debugger Architecture, *Proceedings of the Fifth IEEE International Conference on Mobile Agents (MA'2001)*, Atlanta, Georgia, USA, December 2-4, 2001.
http://cia.informatik.uni-ulm.de/

[17] Jul, E., Steensgaard, B., Object and Native Code Thread Mobility among Heterogeneous Computers, *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'15)*, Copper Mountain Resort, Colorado, USA, December 1995.
http://www-ece.rice.edu/SOSP15/paper-list.html

[18] Lindholm, T., Yellin, F., *The Java Virtual Machine Specification (2nd Edition)*, Addison Wesley, 1999.

[19] Meloan, S., *The Java HotSpot Performance Engine: An In-Depth Look*, Sun Microsystems, June 1999.
http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/

[20] Picco, G. P., Mobile Agents, *Proceedings of the 5th IEEE International Conference on Mobile Agents (MA'2001)*, Lecture Notes in Computer Science, Volume 2240, Atlanta, Georgia, USA, December 2-4, 2001.

[21] Pozo, R., Miller, B., *SciMark 2.0 documentation*, 2000.
http://math.nist.gov/scimark2/

[22] Sakamoto, T., Sekiguchi, T., and Yonezawa, A., Bytecode Transformation for Portable Thread Migration in Java, *Proceedings of the Fourth International Symposium on Mobile Agents 2000 (MA'2000)*, Zürich, Switzerland, September 13-15, 2000.
http://www.yl.is.s.u-tokyo.ac.jp/~takas/

[23] Sekiguchi, T., Masuhara, H., Yonezawa, A., A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation, *Coordination Languages and Models, Lecture Notes in Computer Science*, Volume 1594, April 1999. http://web.yl.is.s.u-tokyo.ac.jp/amo/

[24] Suezawa, T., Persistent Execution State of a Java Virtual Machine, *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, California, USA, June 2000. http://www.ifi.unizh.ch/~suezawa/

[25] Sun Microsystems, *Java 2 Platform, Standard Edition, v 1.3.1 - API Specification*, Sun Microsystems, 2002. http://java.sun.com/j2se/1.3/docs/api/

[26] Sun Microsystems, *The Source for Java Technology*, Sun Microsystems, 2002. http://java.sun.com/

[27] Sun Microsystems, *Java JIT Compiler Overview*, Sun Microsystems, 2002. http://www.sun.com/solaris/jit/

[28] Sun Microsystems, *Java Virtual Machine Debug Interface Reference*, Sun Microsystems, 2002. http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/jvmdi.html

[29] Sun Microsystems, *Applets*, Sun Microsystems, 2002. http://java.sun.com/applets/

[30] Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., and Verbaeten, P., Portable Support for Transparent Thread Migration in Java, *Proceedings of the Fourth International Symposium on Mobile Agents 2000 (MA'2000)*, Zürich, Switzerland, September 13-15, 2000. http://www.cs.kuleuven.ac.be/ eddy/research.html

[31] Wong, D., Paciorek, N., Walsh, T., DiCelie, J, Young, M., Peet, B., Concordia: An Infrastructure for Collaborating Mobile Agents, *First International Workshop on Mobile Agents (MA'97)*, Berlin, Germany, April 1997. http://www.concordiaagents.com/

Contents

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803